JPA and Annotations: Examples

Estimated time: 5 min

Learning objectives

At the end of this reading, you will be able to:

• Explain Java Persistence API (JPA)
• Apply important annotations including @Entity, @Id, @GeneratedValue, and @Transient

Introduction

Java Persistence API (JPA) is the standard API for object-relational mapping (ORM) in Java. It allows developers to map Java classes to database tables, so you can interact with relational data using simple Java objects, no SQL required.

JPA is just a specification, so it requires implementation. In most Spring Boot projects, the default implementation is Hibernate.

With JPA, instead of manually writing `INSERT`, `UPDATE`, or `SELECT` queries, you create annotated classes like `Doctor`, `Patient`, and `Appointment`, and Spring Data JPA handles the rest.

Why data validation matters

Before data reaches the database, and long before it is used in business logic, you want to make sure it is valid, complete, and safe. Validation rules enforce:

- Required fields
  - Example: Username must not be null
- Format rules
  - Example: Email must be valid
- Length restrictions
  - Example: Name must be 3–100 characters
- Logical constraints
  - Example: Date must be in the future

This protects your app from bad data, reduces bugs, and improves user experience.

Spring Boot uses Hibernate Validator, the reference implementation of the Bean Validation API (`javax.validation` or `jakarta.validation`), to enforce these rules via annotations.

Basic annotations

Annotation	Purpose
`@Entity`	Declares the class as a database-mapped entity
`@Id`	Marks the primary key
`@GeneratedValue`	Auto-generates the primary key
`@NotNull`	Field must not be null
`@Size(min, max)`	String length constraint
`@Email`	Field must be a valid email address
`@Pattern(regex)`	Field must match the given regular expression
`@JsonProperty(WRITE_ONLY)`	Hides sensitive fields (passwords) from JSON output
`@ManyToOne`	Defines a many-to-one relationship
`@Future`	Date/time must be in the future
`@Transient`	Field should not be persisted in the database

Example 1: Admin model

This simple model demonstrates how to define an entity and protect sensitive data:

```
java
@Entity
public class Admin {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @NotNull(message = "Username cannot be null")
    private String username;
    @NotNull
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private String password;
    // Getters and setters
}
```

- `@Entity` maps this class to a database table
- `@JsonProperty(WRITE_ONLY)` ensures that the password is never exposed in API responses
- `@NotNull` ensures that required fields are not left empty

Example 2: Doctor with validation

```
java
@Entity
public class Doctor {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @NotNull
    @Size(min = 3, max = 100)
    private String name;
    @NotNull
    @Size(min = 3, max = 50)
    private String specialty;
    @Email
    @NotNull
    private String email;
    @Size(min = 6)
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private String password;
    @Pattern(regexp = "\\d{10}")
    private String phone;
    @ElementCollection
    private List<String> availableTimes;
}
```

- Field-level validation ensures data quality
- `@Email`, `@Size`, and `@Pattern` enforce structure
- `@ElementCollection` is used to store a list of simple values, such as strings, in a separate table

Example 3: Appointment with relationships and transient logic

```
java
@Entity
public class Appointment {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @ManyToOne
    @NotNull
    private Doctor doctor;
    @ManyToOne
    @NotNull
    private Patient patient;
    @Future
    private LocalDateTime appointmentTime;
    private int status; // 0 = Scheduled, 1 = Completed
    @Transient
    public LocalDateTime getEndTime() {
        return appointmentTime.plusHours(1);
    }
}
```

- `@ManyToOne` defines the relationship between appointment and both doctor and patient
- `@Future` enforces logical constraints on scheduling
- `@Transient` prevents `getEndTime()` from being stored in the database

MongoDB support: Using @Document

While JPA works with relational databases, Spring Boot also supports NoSQL databases like MongoDB via Spring Data MongoDB. Instead of `@Entity`, you use `@Document`.

```
java
@Document(collection = "prescriptions")
```

```
public class Prescription {
    @Id
    private String id;
    @NotNull
    @Size(min = 3, max = 100)
    private String patientName;
    @NotNull
    private Long appointmentId;
    @NotNull
    @Size(min = 3, max = 100)
    private String medication;
    @Size(max = 200)
    private String doctorNotes;
}
```

- MongoDB stores this as a document in a JSON-like format
- You still use the same validation annotations as JPA models

---

Validation and serialization security

When exposing data through a REST API, it's critical to secure sensitive fields like passwords. Use:
java
@JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
private String password;

This allows the field to be **set on POST** but **never returned** in a GET response.

You can also use `@JsonIgnore` to completely exclude internal fields from being serialized to JSON.

Summary

In this reading, you learned that:

- Creating and validating model classes:
  - Map entities to relational tables and MongoDB documents
  - Define entity relationships like `@ManyToOne`
  - Validate user input using annotations like `@NotNull`, `@Email`, and `@Pattern`
  - Exclude logic fields using `@Transient`
  - Secure sensitive fields using JSON serialization annotations
- These patterns form the backbone of real-world Java backend applications, clean, secure, and consistent data models.

Author(s)

Upkar Lidder
IBM Skills Network Team